

# A Parallel Algorithm for Exact Bayesian Structure Discovery in Bayesian Networks

**Yetian Chen**

YETIANC@IASTATE.EDU

**Jin Tian**

JTIAN@IASTATE.EDU

*Department of Computer Science*

*Iowa State University*

*Ames, IA 50011, USA*

**Olga Nikolova**

OLGA.NIKOLOVA@SAGEBASE.ORG

*Sage Bionetworks*

*Seattle, WA, USA*

**Srinivas Aluru**

ALURU@CC.GATECH.EDU

*School of Computational Science and Engineering*

*College of Computing, Georgia Institute of Technology*

*Atlanta, GA 30332, USA*

**Editor:**

## Abstract

Exact Bayesian structure discovery in Bayesian networks requires exponential time and space. Using dynamic programming (DP), the fastest known serial algorithm computes the exact posterior probabilities of structural features in  $O(n2^n)$  time and space, if the number of parents per node or *indegree* is bounded by a constant  $d$ . Here we present a parallel algorithm capable of computing the exact posterior probabilities for all  $n(n-1)$  edges with optimal parallel time and space efficiency. That is, if  $p = 2^k$  processors are used, the run-time and space usage reduce to  $O(n2^{n-k} + k(n-k)^d)$  and  $O(n2^{n-k})$ , respectively. Our algorithm is based on the observation that the original DP steps constitute a  $n$ -D hypercube. In our algorithm, we take a delicate way to coordinate the computations of correlated DP procedures such that large amount of data exchange is suppressed. Further, we develop parallel techniques for two variants of the well-known *zeta transform*, which have applications outside the context of Bayesian networks. We demonstrate the capability of our algorithm on datasets with up to 33 variables and its scalability on up to 2048 processors.

**Keywords:** Parallel Algorithm, Exact Structure Discovery, Bayesian Networks

## 1. Introduction

Bayesian networks (BNs) are probabilistic graphical models that represent a set of random variables and their conditional dependencies via a directed acyclic graph (DAG). Learning the structures of Bayesian networks from data has been the major concern in many applications of BNs. In one of the applications, one aims to find a BN that best explains the observations and then utilizes this optimal BN for predictions or inferences. In another one, we are interested in finding the local structural features that are highly probable. In causal discovery, for example, one aims at the identification of (direct) causal relations among a set of variables, represented by the edges in the network structure (Heckerman et al., 1997).

Among the approaches to address the optimal BN learning problem, score-based structure search method formalizes the problem as an optimization problem where a decomposable score is used to measure the fitness of a DAG to the observed data, then a certain search approach is employed to maximize (or minimize) the score over the space of possible DAGs (Cooper and Herskovits, 1992; Heckerman et al., 1997). While for the application of discovering highly probable structural features, Bayesian methods are extensively used. In these methods, one provides a prior probability distribution  $P(G)$  over the space of possible Bayesian networks and then computes the posterior distribution  $P(G|D)$  of the network structure  $G$  given data  $D$ . One can then compute the posterior probability of any structural features by averaging over all possible networks. Regardless of the applications, the exact structural learning problems are considered hard since the number of possible networks is super-exponential  $O(n! 2^{n(n-1)/2})$  in the number of variables  $n$ . Indeed, it has been showed in (Chickering et al., 1995) that finding an optimal Bayesian network structure is NP-hard even when the maximum in-degree is bounded by a constant greater than one.

The fastest known solutions to optimal BN learning problem are a set of dynamic programming (DP) algorithms that run in time and space of  $O(n2^n)$  (Ott et al., 2004; Koivisto and Sood, 2004; Singh and Moore, 2005; Silander and Myllymäki, 2006). Likewise, the posterior probability of structural features can be computed by analogous DP techniques. For example, the algorithms developed in (Koivisto and Sood, 2004) and (Koivisto, 2006) can compute the exact marginal posterior probability of any subnetwork (e.g., an edge) and the exact posterior probabilities for all  $n(n-1)$  potential edges in  $O(n2^n)$  time and space, assuming that the *indegree*, i.e., the number of parents of each node, is bounded by a constant. However, these algorithms require a special form of the structural prior  $P(G)$ , which deviates from the simplest uniform distribution and does not respect Markov equivalence (Friedman and Koller, 2003). If adhering to the uniform distribution, the fastest known algorithm is substantially slower, taking time  $O(3^n)$  and space  $O(n2^n)$  (Tian and He, 2009). Due to their exponential time and space complexities, the largest networks these DP algorithms can solve on a typical desktop computer with a few GBs of memory do not exceed 25 variables.

While both the time and space requirements grow exponentially as  $n$  increases, it is the space requirement being the bottleneck in practice. By noting this, several techniques have been developed to reduce the space usage. In (Malone et al., 2011), the original DP algorithm for finding optimal BNs is improved such that only the scores and information for two adjacent layers in the recursive graph are kept in memory at once in each DP loop. This manipulation reduces the memory requirements to  $O(\binom{n}{n/2})$ . And they showed the implementation of the algorithm solved a problem of 30 variables in about 22 hours using 16 GB memory. However, the implementation of their algorithm needs external memory (i.e., hard disk) to store the entire recursive graph. This may slow down the algorithm due to the slow access to hard disk. Further, the algorithm is not scalable on larger problems as the  $O(\binom{n}{n/2})$  space requirement still grows very fast as  $n$  increases. As another method to reduce the space usage, (Parviainen and Koivisto, 2009) proposed several schemes to trade space against time. If little space is available, a divide-and-conquer scheme recursively splits the problem to subproblems, each of which can be solved independently. This scheme results in time  $2^{2n-s}n^{O(1)}$  in space  $2^s n^{O(1)}$  for any  $s = n/2, n/4, n/8, \dots$ , where  $s$  is the size of the subproblems. If moderate amounts of space are available, a pairwise scheme splits

the search space by fixing a class of partial orders on the set of variables. This manipulation yields run-time  $2^n(3/2)^p n^{O(1)}$  in space  $2^n(3/4)^p n^{O(1)}$  for any  $p = 0, 1, \dots, n/2$  where  $p$  is a parameter controlling the space-time trade-off. Although both schemes make it practical to solve larger problems using limited space, they make a huge sacrifice in run-time.

Recently, parallel computing has drawn increasing attention from AI community due to its ability to solve computationally intensive problems that cannot be solved by serial computing in a reasonable time. Further, if the problem can be splitted to subproblems that can be solved independently with much less space, it is well suited to be solved by using computing clusters with distributed memory. As a result, several parallel algorithms have already been developed for the optimal BN learning problem. First, as mentioned by the authors, the pairwise scheme proposed in (Parviainen and Koivisto, 2010) allows easy parallelization on up to  $2^p$  processors for any  $p = 0, 1, \dots, n/2$ . Each of the processors solves a subproblem independently in time  $2^n(3/4)^p n^{O(1)}$  in space  $2^n(3/4)^p n^{O(1)}$ . Compared to the serial algorithm that runs in time and space of  $2^n n^{O(1)}$ , the parallel efficiency is  $(2/3)^p$ , which is suboptimal. Further, they only implemented the pairwise scheme to compute only the subproblems. Thus, although their results suggest the implementation is feasible up to around 31 variables, their estimate ignores the parallelization overhead that generally becomes problematic in parallelization. (Tamada et al., 2011) presented a parallel algorithm that takes a different way to split the search space so that the required communication between subproblems is minimal. The overall time and space complexity is  $O(n^{\sigma+1}2^n)$ , where  $\sigma = 0, 1, \dots, > 0$  controls the communication-space trade-off. This algorithm, as mentioned, has slightly greater space and time complexities than the algorithm in (Parviainen and Koivisto, 2009) because of redundant calculations of DP steps. Their implementation of the algorithm was able to solve 32-node network in about 5 days 14 hours using 256 processors with 3.3 GB memory per processor. However, it didn't scale well on more than 512 processors as the parallel efficiency decreased significantly from 0.74 on 256 processors to 0.39 on 1024 processors. This poses a barrier to solving larger problems by utilizing larger computing clusters. (Nikolova et al., 2009) and (Nikolova et al., 2013) described a novel parallel algorithm that realizes direct parallelization of the original DP algorithm with optimal parallel efficiency. This algorithm is based on the observation that the DP steps constitute a lattice equivalent to an  $n$ -dimensional ( $n$ -D) hypercube, which has been proved to be a very powerful interconnection network topology used by most of modern parallel computer systems (Dally and Towles, 2004; Ananth et al., 2003; Loh et al., 2005). In the lattice formed by the DP steps, data exchange only happens between two adjacent nodes. On the other hand, in a hypercube interconnection network, the neighbors communicate with each other much more efficiently than other pairs of nodes. By noting this, the parallel algorithm takes a direct mapping of the DP steps to the nodes of a hypercube, thus is communication-efficient. Further, this hypercube algorithm does not calculate redundant steps or scores. These two features render the implementation of the algorithm scalable on up to 2048 processors (Nikolova et al., 2013). Using 1024 processors with 512 MB memory per processor, they solved a problem with 30 variables in 1.5 hours.

Compared to the optimal BN learning problem, application of parallelism in discovering high-probable structural features has not been studied so extensively. To our knowledge, there are no specific parallel algorithms developed for computing the exact posterior probability of structural features. Although (Parviainen and Koivisto, 2010) extended the

parallelizable partial-order scheme to the latter problem, they didn't offer any explicit way to parallelize it. Although the DP techniques for these two problems are analogous, they differ in some significant places. These differences prohibit the direct adaption of the existing parallel algorithms for the former problem to the latter one. First, the DP algorithm for optimal BN learning involves only one DP procedure. All scores for a certain node set are computed in one DP step, therefore can be computed on one processor. Thus, the mapping of node sets to processors is very straightforward. However, the DP algorithm for computing the posterior probability of structural features involves several separate DP procedures, responsible for computing different scores. These DP procedures, though can be performed separately, rely on the completion of one another. Thus, it is a challenge to effectively coordinate the computations of these DP procedures in a parallel setup. Failure to do this may greatly harm the parallel efficiency. Second, the DP algorithm for the latter problem involves two critical subtasks, each of which calls for a fast computation of a zeta transform variant. These two zeta transform variants require efficient parallel processing.

To fill up the gap, we develop a parallel algorithm to compute the exact posterior probability of substructures (e.g., edges) in Bayesian networks. Our algorithm realizes direct parallelization of the DP algorithm in (Koivisto, 2006) with nearly perfect load-balancing and optimal parallel time and space efficiency, i.e., the time and space complexity per processor are  $O(n2^{n-k})$  respectively, for  $p$  number of processors, where  $k = \log(p)$ . We shall say that our parallel algorithm is an extension of Nikolova *et al.*'s hypercube algorithm to the substructure discovery problem. However, because of the difficulties discussed previously, our work goes beyond that by a significant margin. First, we adopt a delicate way to map the calculation of various scores to the processors such that large amount of data exchange between non-neighboring processors is avoided during the transition among the separate DP procedures. This manipulation significantly reduces the time spent in communication. Second, we develop novel parallel algorithms for two fast zeta transform variants. As zeta transforms are fundamental objects in combinatorics and algorithmics, the parallel algorithms developed here would also benefit the researches outside the context of Bayesian networks (Björklund *et al.*, 2007, 2010; Nederlof, 2009).

The rest of the paper is organized as follows. In Section 2, we present some preliminaries of exact structure discovery in BNs and briefly review Koivisto *et al.*'s DP algorithm, upon which our parallel algorithm is based. In Section 3, we present our parallel algorithm for computing the posterior probability of structural features and conduct a theoretical analysis on its run-time and space complexity. In Section 4, we empirically demonstrate the capability of our algorithm on a Dell PowerEdge C8220 cluster. Discussions and conclusions are presented in Section 5.

## 2. Exact Bayesian Structure Discovery in Bayesian Networks

Formally, a Bayesian network is a DAG that encodes a joint probability distribution over a vector of random variables  $x = (x_1, \dots, x_n)$  with each node of the graph representing a variable in  $x$ . For convenience we will typically work on the index set  $V = \{1, \dots, n\}$  and represent a variable  $x_i$  by its index  $i$ . The DAG is represented as a vector  $G = (G_1, \dots, G_n)$  where each  $G_i$  is a subset of the index set  $V$  and specifies the parents of  $i$  in the graph.

Given a set of observations  $D$ , in the Bayesian approach to learn Bayesian networks from the observations, we compute the posterior probability of a network  $G$  as

$$P(G|D) = P(D|G)P(G)/P(D). \quad (1)$$

## 2.1 Computing Posteriors of Structural Features

A structural feature, e.g., an edge, is conveniently represented by an indicator function such that  $f(G)$  is 1 if the feature is present in  $G$  and 0 otherwise. The posterior probability of any structural feature can be computed by averaging over all possible networks:

$$P(f|D) = \sum_G f(G)P(G|D). \quad (2)$$

Instead of directly summing over the super-exponential DAG space, (Friedman and Koller, 2003) proposed to work on the order space, which was demonstrated more efficient and convenient. Formally, an order  $\prec$  is a linear order on the index set  $V$ , represented as a vector  $(L_1, \dots, L_n)$ , where  $L_i$  specifies the predecessors of  $i$  in the order, i.e.,  $L_i = \{j : j \prec i\}$ . We say that a structure  $G = (G_1, \dots, G_n)$  is consistent with an order  $L_i = \{j : j \prec i\}$  if  $G_i \subseteq L_i$  for all  $i$ . With this definition, we can rewrite Eq.(2) as

$$P(f|D) = P(f, D)/P(D), \quad (3)$$

where

$$P(f, D) = \sum_{\prec} P(\prec)P(f, D|\prec). \quad (4)$$

Assuming the model is modular over  $\prec$ , if  $G$  is consistent with  $\prec$ , then

$$P(\prec, G) = \prod_{i=1}^n q_i(L_i)\rho_i(G_i), \quad (5)$$

where each  $q_i(L_i)$  and  $\rho_i(G_i)$  is some function from the subsets of  $V - \{i\}$  to the non-negative reals.

We will also make the standard assumptions on structural modularity, global and local parameter independence, and parameter modularity (Cooper and Herskovits, 1992). Further, in this paper, we consider only modular features, i.e.,  $f(G) = \prod_{i=1}^n f_i(G_i)$ , where each  $f_i(G_i)$  is an indicator function with values either 0 or 1. For example, an edge  $u \rightarrow v$  can be represented by setting  $f_v(G_v) = 1$  if and only if  $u \in G_v$ , and setting  $f_i(G_i) = 1$  for all  $i \neq v$ . In addition, we assume the number of parents of each node is bounded by a constant  $d$ . With these assumptions, we can factorize Eq.(4) as

$$P(f, D) = \sum_{\prec} \prod_{i=1}^n q_i(L_i) \sum_{G_i \subseteq L_i: |G_i| \leq d} \rho_i(G_i) p(x_i|x_{G_i}, G_i) f_i(G_i), \quad (6)$$

where  $p(x_i|x_{G_i}, G_i)$  is the local marginal likelihood for variable  $i$ . For convenience, for each family  $(i, G_i)$ ,  $i \in V$ ,  $G_i \subseteq V - \{i\}$ , we define the local score

$$B_i(G_i) \equiv \rho_i(G_i)p(x_i|x_{G_i}, G_i)f_i(G_i), \quad (7)$$

which measures the local goodness of  $G_i$  as the parents of  $i$ . Note that if we assume the bounded indegree  $d$ , we only need to compute  $B_i(G_i)$  for  $G_i \subseteq V - \{i\}$  with  $|G_i| \leq d$ .

Further, for all  $i \in V$ ,  $L_i \subseteq V - \{i\}$ , define

$$A_i(L_i) \equiv q_i(L_i) \sum_{G_i \subseteq L_i: |G_i| \leq d} B_i(G_i). \quad (8)$$

The sum on the right is known as a variant of the *zeta transform* of  $B_i$ , evaluated at  $L_i$ . Now Eq.(6) can be neatly written as

$$P(f, D) = \sum_{\prec} \prod_{i=1}^n A_i(L_i) = F(V), \quad (9)$$

where the function  $F$ , read as forward sum, is defined for all  $S \subseteq V$  by setting  $L(\emptyset) \equiv 1$  and, recursively,

$$F(S) \equiv \sum_{i \in S} A_i(S - \{i\})F(S - \{i\}). \quad (10)$$

With this definition,  $P(f, D) = F(V)$  can be computed effectively with dynamic programming. Then the posterior probability of the feature  $f$  is obtained as  $P(f|D) = P(f, D)/P(D)$ , where  $P(D)$  can be computed like  $P(f, D)$  by simply setting all features  $f_i(G_i) = 1$ , i.e.  $P(f = 1, D) = P(D)$ .

Computing  $B_i(G_i)$  scores for all  $i \in V$ ,  $|G_i| \leq d$  takes  $O(n^{d+1})$  time.<sup>1</sup> All  $A_i$  scores can be computed in  $O(n2^n)$  time with a technique called the *fast truncated upward zeta transform*<sup>2</sup> (Koivisto and Sood, 2004). The recursive computation of  $F(V)$  takes  $O(\sum_{i=0}^n i \cdot \binom{n}{i}) = O(n2^n)$  time. The total time for computing one feature (i.e., an edge) is therefore  $O(n2^n)$ .

## 2.2 Computing Posterior Probabilities for All Edges

If the application is to compute the posteriors for all  $n(n-1)$  edges, we can run above algorithm separately for each edge. Then the time for computing all  $n(n-1)$  edges is  $O(n^3 2^n)$ . Since the computations for different edges involve a large proportion of overlapping elements, a forward-backward algorithm was provided in (Koivisto, 2006) to reduce the time to  $O(n2^n)$ . For all  $S \subseteq V$ , define a “backward function” recursively as

$$R(S) = \sum_{i \in S} A_i(V - S)R(S - \{i\}), \quad (11)$$

and  $R(\emptyset) = 1$ . With the definition of  $R$ , for any fixed node  $v \in V$  (the endpoint of an edge),  $u \in V - \{v\}$ , the joint distribution  $P(u \rightarrow v, D)$  can be expressed as

- 
1. We assume the computation of  $p(x_i|x_{G_i}, G_i)$  takes  $O(1)$  time here. However, it is usually proportional to the sample size  $m$ .
  2. It is called Möbius transform in (Koivisto and Sood, 2004; Koivisto, 2006), but *zeta transform* is actually the correct term.

$$P(u \rightarrow v, D) = \sum_{u \in G_v \subseteq V - \{v\}: |G_v| \leq d} B_v(G_v) \Gamma_v(G_v), \quad (12)$$

where

$$\Gamma_v(G_v) \equiv \sum_{G_v \subseteq S \subseteq V - \{v\}} q_v(S) F(S) R(V - \{v\} - S). \quad (13)$$

Again, the sum on the right of Eq.(13) is another variant of *zeta transform*. Provided that  $B_i, A_i, F, R$  are precomputed with respect to  $f \equiv 1$ , for any endpoint node  $v$ ,  $\Gamma_v(G_v)$  can be computed in  $O(2^n)$  time for all  $G_v \subseteq V - \{v\}$ ,  $|G_v| \leq d$  with a technique called *fast downward zeta transform* (Koivisto, 2006). To evaluate Eq.(12) for different  $u \in V - \{v\}$ , we only need to recompute the function  $B_v$ . This corresponds to changing just the function  $f_v$ . Thus, evaluating Eq.(12) takes  $O(n^d)$  time.

We then arrived at the following algorithm for computing the posteriors for all  $n(n-1)$  edges. Let the functions  $B_i, A_i, \Gamma_i, F$  and  $R$  be defined with respect to the trivial feature  $f \equiv 1$ .

---

**Algorithm 1** Compute posterior probabilities for all  $n(n-1)$  edges by DP (Koivisto, 2006)

---

- 1: **for** all  $i \in V$  and  $G_i \subseteq V - \{i\}$  with  $|G_i| \leq d$ : compute  $B_i(G_i)$ .
  - 2: **for** all  $i \in V$  and  $L_i \subseteq V - \{i\}$ : compute  $A_i(L_i)$ .
  - 3: **for** all  $S \subseteq V$ : compute  $F(S)$  recursively.
  - 4: **for** all  $S \subseteq V$ : compute  $R(S)$  recursively.
  - 5: **for** all  $v \in V$  **do**
  - 6:     **for** all  $G_v \subseteq V - \{v\}$  with  $|G_v| \leq d$ : compute  $\Gamma_v(G_v)$ .
  - 7:     **for** all  $u \in V - \{v\}$  **do**
  - 8:         Compute  $P(u \rightarrow v, D) = \sum_{u \in G_v \subseteq V - \{v\}: |G_v| \leq d} B_v(G_v) \Gamma_v(G_v)$
  - 9:         Evaluate  $P(u \rightarrow v | D) = P(u \rightarrow v, D) / F(V)$ .
  - 10:     **end for**
  - 11: **end for**
- 

Adding up the time for all steps, the total computation time for evaluating all  $n(n-1)$  edges is  $O(n2^n)$ .

### 3. Parallel Algorithm

The algorithm presented in Section 2 serves as a good base for parallelization. Although it is more complicated than the DP algorithms for finding optimal BN, they do share some features. Specifically, the recursive computations of functions  $F$  and  $R$  over node sets  $S \subseteq V$  are analogous to DP steps in those algorithms for finding optimal BN. Thus, the parallel techniques developed for the latter problem can be used. In our algorithm, we adapt Nikolova *et al.*'s hypercube algorithm and use it as sub-routines to compute functions  $F$  and  $R$ . However, as discussed previously, some difficulties prohibit the direct adaption. As showed in **Algorithm 1**, the computation consists of several consecutive procedures, each

of which is responsible for computing a particular function. The computations of these functions depend on one another. For example, computing  $A_i$ 's needs  $B_i$ 's having been computed; computing  $F$  and  $R$  need  $A_i$ 's being available;... etc. Note that all functions are evaluated over  $2^n$  of subsets  $S \subseteq V$ . In the hypercube algorithm, these subsets are computed in different processors, thus stored distributedly. Generally, processors need exchange the required scores for computing a new function. Thus, it is challenging to coordinate the computations of these functions to avoid large amount of data exchange among processors, particularly those non-neighboring processors. In our algorithm, we adopt a delicate way to achieve this. Further, functions  $A$  and  $\Gamma$  correspond respectively to two variants of *zeta transform* which are computationally intensive. Here we design novel parallel algorithms for them. Finally, we integrate these techniques into a parallel algorithm capable of computing the posteriors  $P(u \rightarrow v|D)$  for all  $n(n-1)$  edges with nearly perfect load-balancing and optimal parallel efficiency.

To facilitate presentation, in Section 3.1, we first describe an ideal case, i.e., mapping the computations to an  $n$ -dimensional hypercube computing cluster, where  $n$  is the number of variables in domain. In Section 3.2, we then generalize the mapping to a  $k$ -dimensional hypercube with  $k < n$ .

### 3.1 $n$ -D Hypercube Algorithm

In Section 3.1.1, we first describe the parallel algorithms for computing functions  $F$  and  $R$  as they explain why we base our parallel algorithm on the hypercube model. We postpone the discussion of computing  $B_i$ ,  $A_i$  scores to Section 3.1.2.

#### 3.1.1 COMPUTING $F(S)$ AND $R(S)$

The DP algorithm for computing functions  $F$  can be visualized as operating on the lattice  $\mathcal{L}$  formed by the partial order “set inclusion” on the power set of  $V$  (see Figure 1). The lattice  $\mathcal{L}$  is a directed graph  $(V', E')$ , where  $V' = 2^V$  and  $(T, S) \in E'$  if  $T \subset S$  and  $|S| = |T| + 1$ . The lattice is naturally partitioned into levels, where level  $l \in [0, n]$  contains all subsets of size  $l$ . A node  $S$  at level  $l$  has  $l$  incoming edges from nodes  $S - \{i\}$  for each  $i \in S$ , and  $n - l$  outgoing edges to nodes  $S \cup \{j\}$  for each  $j \notin S$ . By Eq.(10), node  $S$  receives  $A_i(S - \{i\}) \cdot F(S - \{i\})$  from each of its incoming edge and computes  $F(S)$  by summing over  $l$  such scores. Assuming  $A_j(S)$  for all  $j \notin S$  are precomputed and available at node  $S$ , node  $S$  will compute  $A_j(S) \cdot F(S)$  for all  $j \notin S$  then send the scores to corresponding nodes. For example,  $A_j(S) \cdot F(S)$  is sent to node  $S \cup \{j\}$  so that it can be used for computing  $F(S \cup \{j\})$ . Each level in the lattice can be computed concurrently, with data flowing from one level to the next.

If each node in  $\mathcal{L}$  is mapped to a processor in a computer cluster, the undirected version of  $\mathcal{L}$  is equivalent to an  $n$ -dimensional ( $n$ -D) hypercube. We encode a subset  $S$  by an  $n$ -bit string  $\omega$ , where  $\omega[i] = 1$  if variable  $i \in S$  and  $\omega[i] = 0$  otherwise. As lattice edges connect pairs of nodes that differ in the existence of one element, they naturally correspond to hypercube edges (Figure 1). This suggests an obvious parallelization on an  $n$ -D hypercube.

The  $n$ -D hypercube algorithm runs in  $n + 1$  steps. Let  $\omega$  denote the id of a processor and let  $\mu(\omega)$  denote the number of 1's in  $\omega$ . Each processor is active in only one time step – processor  $\omega$  is active in time step  $\mu(\omega)$ . It receives one  $A_i(S - \{i\}) \cdot F(S - \{i\})$  value from



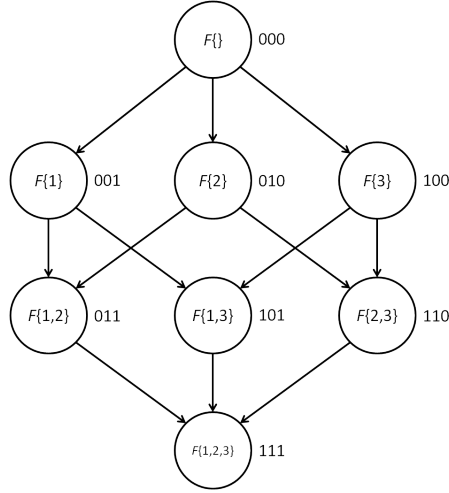


Figure 1: A lattice for a domain of size 3. The binary string labels on the right-hand side of each node show the correspondence with a 3-dimensional hypercube.  $B_i$ ,  $A_i$  and function  $F$  for each subset  $S$  are computed at corresponding processor.

each of  $\mu(\omega)$  neighbors obtained by inverting one of its 1 bits to 0. It then computes its  $F(S)$  function, computes  $A_j(S) \cdot F(S)$  for all  $j \notin S$  and sends them to its  $n - \mu(\omega)$  neighbors obtained by inverting one of its 0 bits to 1. The run-time of step  $l$  is  $O(l + n - l) = O(n)$ . The parallel run-time for computing all  $F$  scores is  $O(n^2)$  in total.

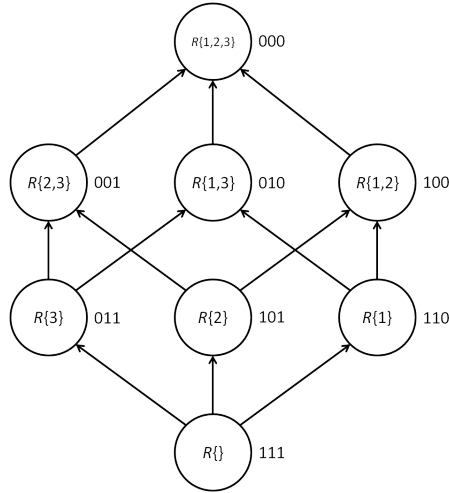


Figure 2: Map the computation of function  $R(S)$  to the  $n$ -D hypercube. The binary string label on the right-hand side of each node denote the id of the processor.

We can parallelize the computation of function  $R$  in the same manner. However, we have assumed  $A_j(S)$  for all  $j \notin S$  are available only at node  $S$ . To compute  $R(S)$ , node  $S$  need receive  $A_i(V - S) \cdot R(S - \{i\})$  from its neighbors. However,  $A_i(V - S)$  are available at neither node  $S - \{i\}$  nor node  $S$ , but node  $V - S$ . Further, it is not a good idea either to have

$F(S)$  and  $R(S)$  at the same processor as each term of the summation in the computation of  $\Gamma$  scores requires different  $F$  and  $R$  (see Eq. (13)). To avoid large amount of data exchange, we take a completely different mapping for computing  $R$ . The new mapping is illustrated in Figure 2. Note that  $R(S)$  is computed at the processor where  $F(V - S)$  is computed. Processor  $\omega$  receives one  $R(S - \{i\})$  from each of its  $n - \mu(\omega)$  neighbors obtained by inverting one of its 0 bits to 1. It then computes  $R(S)$  by Eq.(11) and sends it to all its  $\mu(\omega)$  neighbors obtained by inverting one of its 1 bits to 0. The processors in the hypercube operate in a bottom-up manner, e.g., starting from processor 111 and ending at processor 000. Similarly, the parallel run-time is  $O(n^2)$ .

### 3.1.2 PARALLEL FAST ZETA TRANSFORMS

In Section 3.1.1, we have assumed  $A_i(S)$  for all  $i \notin S$  are precomputed at node  $S$ . For any  $i \in V$ , computing  $A_i(S)$  for any subset  $S \subseteq V - \{i\}$  requires the summation over all subsets of  $S$ . If processors in the hypercube compute the  $A_i$  independently, the processor for computing  $A_i(V - \{i\})$  for all  $i \in V$  takes  $O(n2^{n-1})$  time. This certainly nullifies our effort. In this section, we describe parallel algorithms in which all  $A_i$  (or  $\Gamma_v$ ) scores can be computed on the  $n$ -D hypercube cluster in  $O(n^2)$  time.

First, we give definitions for two variants of the well-known *zeta transform* (Kennes, 1992).

Let  $V = \{1, \dots, n\}$ . Let  $s : 2^V \rightarrow \mathbb{R}$  be a mapping from the subsets of  $V$  onto the real numbers.  $d$  is a positive integer  $d \leq |V|$ .

**Definition 1** (Koivisto and Sood, 2004): A function  $t : 2^V \rightarrow \mathbb{R}$  is the truncated *upward* zeta transform of  $s$  if

$$t(T) = \sum_{S \subseteq T: |S| \leq d} s(S) \text{ for all } T \subseteq V.$$

**Definition 2** (Koivisto, 2006): A function  $t : 2^V \rightarrow \mathbb{R}$  is the truncated *downward* zeta transform of  $s$  if

$$t(T) = \sum_{T \subseteq S \subseteq V} s(S) \text{ for all } T \subseteq V \text{ with } |T| \leq d.$$

It is easy to see that the summation formulas defining the function  $A_i$  for all  $i \in V$  can be viewed as a case of the truncated *upward* zeta transform on a subset lattice. Similarly, the formula to compute  $\Gamma_v(G_v)$  for all  $v \in V$ ,  $G_v \subseteq V - \{v\}$  with  $|G_v| \leq d$  can be viewed as a case of the truncated *downward* zeta transform.

Two techniques introduced in (Koivisto and Sood, 2004) and (Koivisto, 2006) are able to realize both transforms in  $O(d2^n)$  time, respectively. Here, we present the parallel versions of the two algorithms (See **Algorithm 2** and **Algorithm 3**) that run on an  $n$ -D hypercube computer cluster. The proofs of correctness for the serial versions of both algorithms are given in (Koivisto and Sood, 2004) and (Koivisto, 2006), respectively.

By our previous definition, a subset  $S$  is encoded by an  $n$ -bit string  $\omega$ , where  $\omega[i] = 1$  if variable  $i \in S$  and  $\omega[i] = 0$  otherwise. In an  $n$ -D hypercube,  $\omega$  is also used to denote the id of a processor. We can take this natural mapping so that each processor  $\omega$  is responsible for transforming the subset  $\omega_S$ . This forms the basic idea of the two parallel algorithms.

**Algorithm 2** runs for  $n + 1$  iterations. In each iteration, all  $2^n$  processors operate on their  $S \subseteq V$  concurrently (Lines 4 to 13). In iteration  $j$ , before the computation starts, each processor  $\omega$  with  $\omega[j] = 0$  sends its  $t_{j-1}(S)$  to its neighbor  $\omega'$  obtained by inverting its  $\omega[j]$  to 1, i.e.,  $\omega' = \omega \oplus 2^{j-1}$  (Line 3).<sup>3</sup> The neighbor receiving this  $t_{j-1}$  will perform the addition on line 11 in iteration  $j$  if necessary.

---

**Algorithm 2** Parallel Truncated Upward Zeta Transform ( $V, s, d$ ) on  $n$ -D hypercube

---

**Require:** encode each subset  $S \subseteq V$  by an  $n$ -bit string  $\omega$ , where  $\omega[i] = 1$  if variable  $i \in S$  and  $\omega[i] = 0$  otherwise. Subset  $S$  is computed on processor with id  $\omega$ .

```

1: On each processor,  $t_0(S) \leftarrow s(S)$  for all  $S \subseteq V$  s.t.  $|S| \leq d$  and  $t_0(S) \leftarrow 0$  otherwise.
2: for  $j \leftarrow 1$  to  $n$  do
3:   Processor with  $\omega[j] = 0$  send  $t_{j-1}(S)$  to processor  $\omega' = \omega \oplus 2^{j-1}$ .
4:   for each processor  $\omega$  s.t.  $S \subseteq V$  with  $|S \cap \{j+1, \dots, n\}| \leq d$  do
5:      $t_j(S) \leftarrow 0$ 
6:     if  $|S \cap \{j, \dots, n\}| \leq d$  then
7:        $t_j(S) \leftarrow t_{j-1}(S)$ 
8:     end if
9:     if  $j \in S$  then
10:      Retrive  $t_{j-1}(S - \{j\})$  from processor  $\omega' = \omega \oplus 2^{j-1}$ .
11:       $t_j(S) \leftarrow t_j(S) + t_{j-1}(S - \{j\})$ 
12:    end if
13:  end for
14: end for
15: return  $t_n(S)$ 

```

---



---

**Algorithm 3** Parallel Truncated Downward Zeta Transform ( $V, s, d$ ) on  $n$ -D hypercube

---

**Require:** encode each subset  $S \subseteq V$  by an  $n$ -bit string  $\omega$ , where  $\omega[i] = 1$  if variable  $i \in S$  and  $\omega[i] = 0$  otherwise. Subset  $S$  is computed on processor with id  $\omega$ .

```

1: On each processor,  $t_0(S) \leftarrow s(S)$  for all  $S \subseteq V$ .
2: for  $j \leftarrow 1$  to  $n$  do
3:   Processor with  $\omega[j] = 1$  send  $t_{j-1}(S)$  to processor  $\omega' = \omega \oplus 2^{j-1}$ .
4:   for each processor  $\omega$  s.t.  $S \subseteq V$  with  $|S \cap \{1, \dots, j\}| \leq d$  do
5:      $t_j(S) \leftarrow t_{j-1}(S)$ 
6:     if  $j \notin S$  then
7:       Retrive  $t_{j-1}(S \cup \{j\})$  from processor  $\omega' = \omega \oplus 2^{j-1}$ .
8:        $t_j(S) \leftarrow t_j(S) + t_{j-1}(S \cup \{j\})$ 
9:     end if
10:  end for
11: end for
12: return  $t_n(S)$ 

```

---

3.  $\oplus$  stands for the bitwise exclusive or (XOR) between two binary strings.

In **Algorithm 3**, the mapping of the computation of  $S$  to  $n$ - $D$  hypercube is the same as in **Algorithm 2**. The only difference resides in communication. In iteration  $j$ , after the computation starts, each processor  $\omega$  with  $\omega[j] = 1$  sends its  $t_{j-1}$  to its neighbor  $\omega'$  obtained by inverting its  $\omega[j]$  to 0, i.e.,  $\omega' = \omega \oplus 2^{j-1}$ . The neighbor receiving this  $t_{j-1}$  will perform the addition on line 8 in iteration  $j$  if necessary.

In each iteration, all  $S \subseteq V$  are computed concurrently on a  $n$ - $D$  hypercube, the computation times for both parallel algorithms are  $O(n)$ . Further, in both algorithms, the communications happen only among neighboring processors ( $\omega' = \omega \oplus 2^{j-1}$  always produces two binary strings  $\omega, \omega'$  that differ in only one bit). Thus, the two algorithms are communication-efficient.

With **Algorithm 2**,  $A_i$  for all  $i \in V$  can be computed in  $|V| \cdot O(n) = O(n^2)$  time.<sup>4</sup> Each processor  $\omega$  computes and keeps the corresponding  $A_i(S)$  for all  $i \in V$ , which is the assumption we made in Section 3.1.1. Thus, the mapping adopted by the two algorithms is well suited for our algorithm as it avoids large amount of data exchange when the computation transits to next step.

Before we adopt **Algorithm 3** to compute  $\Gamma_v$  for any  $v \in V$ , we shall compute  $q_v(S)F(S)R(V - \{v\} - S)$  on each processor first (See Eq.(13)). However,  $F(S)$  and  $R(V - \{v\} - S)$  are not at the same processor at the time when we have computed functions  $F$  and  $R$ . Fortunately, they are in the processors who are neighbors in the hypercube. Thus, processors with  $R(V - \{v\} - S)$  should send  $R$  score to the processor keeping  $F(S)$  (see Figure 3) before we transform them to  $\Gamma_v$ . With **Algorithm 3**, computing  $\Gamma_v$  for any fixed  $v \in V$  takes  $O(n)$ . The time for computing  $\Gamma_v(G_v)$  for all  $v \in V$  and  $G_v \subseteq V - \{v\}$  is therefore  $|V| \cdot O(n) = O(n^2)$ .

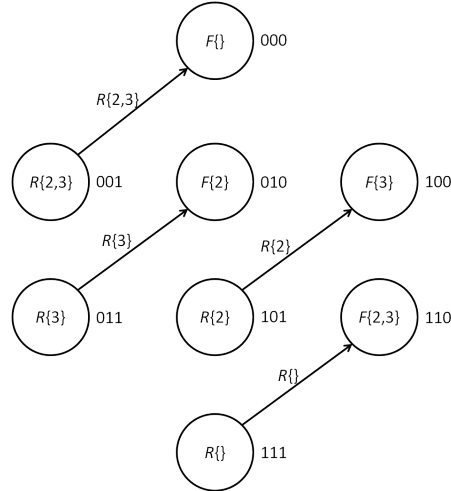


Figure 3: Exchange  $R$  for computing  $\Gamma_v$ . Example shows the case in which  $\Gamma_v$  for  $v = 1$  is computed.

4.  $A_i(S)$  are defined for all  $S \subseteq V - \{i\}$ , instead of  $S \subseteq V$ . However, the algorithm can still be deployed by setting  $A_i(S) = 0$  for all  $S \subseteq V$  s.t.  $i \in S$ .

### 3.1.3 COMPUTING $P(u \rightarrow v|D)$

With  $B_v(G_v)$  and  $\Gamma_v(G_v)$  computed, we can compute  $P(u \rightarrow v, D)$  using Eq. (12). Noting that  $B_v(G_v)$  and  $\Gamma_v(G_v)$  for any  $G_v \subseteq V - \{v\}$  are on the same processor, each processor first computes  $B_v(G_v) \cdot \Gamma_v(G_v)$  locally, then a *MPI\_Reduce*, a collective function in MPI library is executed on the hypercube to compute the sum of  $B_v(G_v) \cdot \Gamma_v(G_v)$  from all processors.  $P(u \rightarrow v, D)$  is then obtained at the processor with highest rank.  $P(u \rightarrow v|D)$  is obtained by evaluating  $P(u \rightarrow v, D)/F(V)$  at the processor with the highest rank. Thus, computing  $P(u \rightarrow v|D)$  for all  $u, v \in V, u \neq v$  takes  $O((\tau + \mu m)n^2)$  time, where  $\tau, \mu, m$  are constants, specifying the latency, bandwidth of the communication network, and the message size.

Adding up the time for each step, the time for evaluating all  $n(n-1)$  edges is  $O(n^2)$ . As the sequential run-time is  $O(n2^n)$ , the parallel efficiency is  $\Theta(1/n)$ .

## 3.2 $k$ -D Hypercube Algorithm

In Section 3.1, we describe the development of our parallel algorithm on an  $n$ -D hypercube. However, we usually expect the number of processors  $p \ll 2^n$ . Let  $p = 2^k$  be the number of processors, where  $k < n$ . We assume that the processors can communicate as in a  $k$ -D hypercube. The strategy is to decompose the  $n$ -D hypercube into  $2^{n-k}$   $k$ -D hypercubes and map each  $k$ -D hypercube to the  $p = 2^k$  processors.

As our previous definition, we use  $S$  to denote the lattice node for subset  $S$  and use  $\omega_S$  to denote the binary string denoting the corresponding hypercube node. We number the positions of a binary string using  $1, \dots, n$  (from right-most bit to left-most bit), and use  $\omega_S[i, j]$  to denote the substring of  $\omega_S$  between and including positions  $i$  and  $j$ . We partition the  $n$ -D hypercube into  $2^{n-k}$   $k$ -D hypercubes based on the left  $n-k$  bits of node id's. For a lattice node  $\omega_S$ ,  $\omega_S[k+1, n]$  specifies the  $k$ -D hypercube it is part of and  $\omega_S[1, k]$  specifies the processor it is assigned to.

### 3.2.1 PARALLEL FAST ZETA TRANSFORMS ON $k$ -D HYPERCUBE

In order to compute  $A$  and  $\Gamma$  functions on a  $k$ -D hypercube, we first generalize the parallel algorithms for the two variants of the *zeta transform* on an  $k$ -D hypercube. We first number the processors in the  $k$ -D hypercube computer cluster with a  $k$ -bit binary string  $r$  such that two adjacent processors  $r, r'$  differ in one bit. The basic idea is, instead of computing the transform for only one subset  $S$ , each processor  $r$  is responsible for computing  $2^{n-k}$  subsets  $S$  such that  $r = \omega_S[1, k]$ . We present the generalized algorithms for the two transforms in **Algorithm 4** and **Algorithm 5**, respectively.

**Algorithm 4** Parallel Truncated Upward Zeta Transform ( $V, s, d$ ) on  $k$ -D hypercube

---

**Require:**  $1 \leq k \leq n$ , each subset  $S \subseteq V$  is encoded by an  $n$ -bit string  $\omega$ , where  $\omega[i] = 1$  if variable  $i \in S$  and  $\omega[i] = 0$  otherwise. Each processor in the  $k$ -D hypercube is encoded by an  $k$ -bit string  $r$ . Subset  $S$  is computed on processor  $r = \omega_S[1, k]$ .

- 1: On each processor,  $t_0(S) \leftarrow s(S)$  for all  $S \subseteq V$  s.t.  $|S| \leq d$  and  $t_0(S) \leftarrow 0$  otherwise.
- 2: **for**  $j \leftarrow 1$  to  $n$  **do**
- 3:   **if**  $j \leq k$  **then**
- 4:     Processor with  $r[j] = 0$  send all its  $t_{j-1}(S)$  s.t.  $j \notin S$  to processor  $r' = r \oplus 2^{j-1}$ .
- 5:   **end if**
- 6:   **for** each  $S \subseteq V$  with  $|S \cap \{j+1, \dots, n\}| \leq d$  on each processor  $r$  **do**
- 7:      $t_j(S) \leftarrow 0$
- 8:     **if**  $|S \cap \{j, \dots, n\}| \leq d$  **then**
- 9:        $t_j(S) \leftarrow t_{j-1}(S)$
- 10:    **end if**
- 11:    **if**  $j \in S$  **then**
- 12:      **if**  $j \leq k$  **then** Retrive  $t_{j-1}(S - \{j\})$  from processor  $r' = r \oplus 2^{j-1}$ .
- 13:      **end if**
- 14:       $t_j(S) \leftarrow t_j(S) + t_{j-1}(S - \{j\})$
- 15:    **end if**
- 16:   **end for**
- 17: **end for**
- 18: **return**  $t_n(S)$

---

**Algorithm 5** Parallel Truncated Downward Zeta Transform ( $V, s, d$ ) on  $k$ -D hypercube

---

**Require:**  $1 \leq k \leq n$ , each subset  $S \subseteq V$  is encoded by an  $n$ -bit string  $\omega$ , where  $\omega[i] = 1$  if variable  $i \in S$  and  $\omega[i] = 0$  otherwise. Each processor in the  $k$ -D hypercube is encoded by an  $k$ -bit string  $r$ . Subset  $S$  is computed on processor  $r = \omega_S[1, k]$ .

- 1: On each processor,  $t_0(S) \leftarrow s(S)$  for all  $S \subseteq V$ .
- 2: **for**  $j \leftarrow 1$  to  $n$  **do**
- 3:   **if**  $j \leq k$  **then**
- 4:     Processor with  $r[j] = 1$  send all its  $t_{j-1}(S)$  s.t.  $j \in S$  to processor  $r' = r \oplus 2^{j-1}$ .
- 5:   **end if**
- 6:   **for** each  $S \subseteq V$  with  $|S \cap \{1, \dots, j\}| \leq d$  on each processor  $r$  **do**
- 7:      $t_j(S) \leftarrow t_{j-1}(S)$
- 8:     **if**  $j \notin S$  **then**
- 9:       **if**  $j \leq k$  **then** Retrive  $t_{j-1}(S \cup \{j\})$  from processor  $r' = r \oplus 2^{j-1}$ .
- 10:      **end if**
- 11:       $t_j(S) \leftarrow t_j(S) + t_{j-1}(S \cup \{j\})$
- 12:    **end if**
- 13:   **end for**
- 14: **end for**
- 15: **return**  $t_n(S)$

---

We now present two theorems that respectively characterize the run-time complexities of the two algorithms.

**Theorem 1** *Parallel Truncated Upward Zeta Transform  $(V, s, d)$  runs in time  $O(d2^{n-k} + k(n-k)^d)$  on  $k$ -D hypercube.* ■

**Proof.** As it is specified, each processor  $r$  computes subsets  $S$  s.t.  $r = \omega_S[1, k]$ . **Algorithm 4** runs for  $n$  iterations. For the iterations  $j = n-d, \dots, n$ , all  $S \subseteq V$  satisfy the condition on line 4, thus each processor performs the computation on line 6–16 for all  $2^{n-k}$  subsets on it. Thus the total computation time for these iterations is  $O((d+1)2^{n-k}) = O(d2^{n-k})$ .

For iterations  $j = 1, \dots, n-d-1$ , the processor with id  $r$  s.t.  $r[i] = 0$  for all  $i = 1, \dots, k$  requires the most time among all processors. Thus, we only need to characterize its run-time, which is proportional to

$$\begin{aligned}
 & \sum_{j=1}^k \sum_{r=0}^d \binom{n-k}{r} + \sum_{j=k+1}^{n-d-1} 2^{j-k} \sum_{r=0}^d \binom{n-j}{r} \\
 &= k \sum_{r=0}^d \binom{n-k}{r} + 2^{-k} \sum_{j=k+1}^{n-d-1} 2^j \sum_{r=0}^d \binom{n-j}{r} \\
 &\leq k \sum_{r=0}^d \binom{n-k}{r} + 2^{-k} \sum_{j=k+1}^{n-d-1} 2^j (n-j)^d \\
 &\leq k \sum_{r=0}^d \binom{n-k}{r} + 2^{-k} \sum_{j=1}^{n-d-1} 2^j (n-j)^d \\
 &\leq k \sum_{r=0}^d \binom{n-k}{r} + 2^{-k} 2^n \sum_{j=0}^{\infty} (1/2)^j j^d
 \end{aligned} \tag{14}$$

The first term  $k \sum_{r=0}^d \binom{n-k}{r} = O(k(n-k)^d)$ . The second term  $2^{-k} 2^n \sum_{j=0}^{\infty} (1/2)^j j^d = O(2^{n-k})$  as the infinite sum converges to a finite limit for a fixed  $d$ . Thus, the time combined for all iteration is  $O(k(n-k)^d) + O(2^{n-k}) + O(d2^{n-k}) = O(d2^{n-k} + k(n-k)^d)$ .

**Theorem 2** *Parallel Truncated Downward Zeta Transform  $(V, s, d)$  runs in time  $O(d2^{n-k})$  on  $k$ -D hypercube.* ■

**Proof.** Each processor  $r$  computes subsets  $S$  s.t.  $r = \omega_S[1, k]$ . In **Algorithm 5**, line 1 takes  $O(2^{n-k})$  time. Lines 2–14 runs for  $n$  iterations. For the iterations  $j = 1, \dots, d$ , all  $S \subseteq V$  satisfy the condition on line 6, thus each processor performs the computation on line 6–13 for all  $2^{n-k}$  subsets on it. Thus the total computation time for these iterations is  $O(d2^{n-k})$ .

For iterations  $j = d+1, \dots, n$ , the processor with id  $r$  s.t.  $r[i] = 0$  for all  $i = 1, \dots, k$  requires the most time among all processors. We characterize its run-time. Its run-time is proportional to

$$\begin{aligned}
& \sum_{j=d+1}^{k+d} 2^{n-k} + \sum_{j=k+d+1}^n 2^{n-j} \sum_{r=0}^d \binom{j-k}{r} \\
&= k2^{n-k} + 2^{-k} \sum_{i=d+1}^{n-k} 2^{n-i} \sum_{r=0}^d \binom{i}{r} \\
&= k2^{n-k} + 2^{-k} \sum_{i=d+1}^{4d-1} 2^{n-i} \sum_{r=0}^d \binom{i}{r} + 2^{-k} \sum_{i=4d}^n 2^{n-i} \sum_{r=0}^d \binom{i}{r} - 2^{-k} \sum_{i=n-k+1}^n 2^{n-i} \sum_{r=0}^d \binom{i}{r} \\
&\leq k2^{n-k} + 2^{-k} \sum_{i=d+1}^{4d-1} 2^n + 2^{-k} \sum_{i=4d}^{n-k} 2^{n-i} \sum_{r=0}^d \binom{i}{r} - 2^{-k} \sum_{i=n-k+1}^n 2^{n-d} \sum_{r=0}^d \binom{d}{r} \\
&= k2^{n-k} + (3d-1)2^{n-k} + 2^{-k} \sum_{i=4d}^{n-k} 2^{n-i} \sum_{r=0}^d \binom{i}{r} - 2^{-k} \sum_{i=n-k+1}^n 2^{n-d} 2^d \\
&= (k+3d-1)2^{n-k} + 2^{-k} \sum_{i=4d}^n 2^{n-i} \sum_{r=0}^d \binom{i}{r} - k2^{n-k} \\
&\leq (3d-1)2^{n-k} + 5 \cdot 2^{n-k}
\end{aligned} \tag{15}$$

The upper bound  $2^{-k} \sum_{i=4d}^n 2^{n-i} \sum_{r=0}^d \binom{i}{r} \leq 5 \cdot 2^{n-k}$  in last step is from **Corollary 3** in (Koivisto, 2006). Thus, the run-time is  $O(d2^{n-k})$ .

### 3.2.2 OVERALL ALGORITHM: PARAREBEL

Now with the  $k$ - $D$  algorithms for the two transforms,  $B_i$ ,  $A_i$  and  $\Gamma_i$  functions can be computed efficiently. As mentioned, each processor with id  $r$  is responsible for computing  $2^{n-k}$  subsets  $S$  such that  $r = \omega_S[1, k]$ .

For computing function  $F$ , we partition the  $n$ - $D$  DP lattice into  $2^{n-k}$   $k$ - $D$  hypercubes based on the left  $n-k$  bits of node id's. For a lattice node  $\omega_S$ ,  $\omega_S[k+1, n]$  specifies the  $k$ - $D$  hypercube it is part of and  $\omega_S[1, k]$  specifies the processor it is assigned to. Using the strategy proposed by (Nikolova et al., 2009), we pipeline the execution of the  $k$ - $D$  hypercubes to complete the parallel execution in  $2^{n-k} + k$  time steps such that all processors are active except for the first  $k$  and last  $k$  time steps during the buildup and finishing off of the pipeline. Specifically, let each  $k$ - $D$  hypercube denoted by an  $(n-k)$  bit string, which is the common prefix to the  $2^k$  lattice/ $k$ - $D$  hypercube nodes that are part of this  $k$ - $D$  sub-hypercube. The  $k$ - $D$  hypercubes are processed in the increasing order of the number of 1's in their bit string specifications, and in lexicographic order within the group of hypercubes with the same number of 1's. Formally, we have the following rule: let  $H_i$  and  $H_j$  be two  $k$ - $D$  hypercubes and let  $S$  and  $T$  bet two nodes in the lattice that map to  $H_i$  and  $H_j$ , respectively. Then, the computation of  $H_i$  is initiated before computation of  $H_j$  if and only if:

1.  $\mu(\omega_S[k+1, n]) < \mu(\omega_T[k+1, n])$ , or
2.  $\mu(\omega_S[k+1, n]) = \mu(\omega_T[k+1, n])$  and  $\omega_S[k+1, n]$  is lexicographically smaller than  $\omega_T[k+1, n]$ .



Figure 4 illustrates a case with  $n = 3$  and  $k = 2$ . In this example, the 3- $D$  lattice is partitioned to two 2- $D$  hypercubes  $H_1$  and  $H_2$ .  $H_1$  is processed before  $H_2$  is processed. One feature of the pipelining is that once a processor completes its computation in one  $k$ - $D$  hypercube, it transits to next  $k$ - $D$  hypercube immediately without waiting for other processors to complete their computations in current hypercube. In Figure 4, for example, once the processor 00 completes node  $\{\}$  and sends out data, it starts on node  $\{3\}$  even processors 01, 10, 01 are still working on their nodes in  $H_1$ . This feature prevents processors from excessive idling during the transitions between consecutive hypercubes.

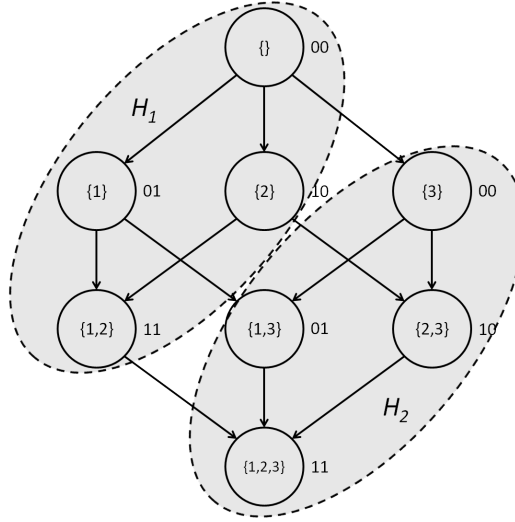


Figure 4: Pipelining execution of hypercubes. Example shows a case with  $n = 3$  and  $k = 2$ .

For computing function  $R(S)$ , the strategy is similar. The only difference is the mapping of the subsets to processors.  $R(S)$  is assigned to the processor with id  $r = \sim \omega_S[1, k]$ <sup>5</sup>, i.e.,  $R(S)$  is computed on the processor where  $F(V - S)$  is computed. In another word, processors operate in the reverse order that they compute  $F(S)$ .

Finally, to compute  $P(u \rightarrow v, D)$  for any  $u, v \in S, u \neq v$ , each processor first adds up all local  $B_v(G_v)\Gamma_v(G_v)$  scores, then a *MPIReduce* is launched on the  $k$ - $D$  hypercube to obtain the sum. The posteriors  $P(u \rightarrow v|D)$  are evaluated as  $P(u \rightarrow v, D)/F(V)$  on the processor with highest rank.

The overall  $k$ - $D$  hypercube algorithm, named as ParaREBEL (*Parallel Rapid Exact Bayesian Edge Learning*), is outlined in **Algorithm 6**.

5.  $\sim \omega_S[1, k]$  denotes the bitwise complement of binary string  $\omega_S[1, k]$ .

---

**Algorithm 6** ParaREBEL( $(V, d, k)$ ) computes the posterior probabilities of all  $n(n-1)$  edges with  $p = 2^k$  processors.

---

**Require:**  $1 \leq k \leq n = |V|$ , each subset  $S \subseteq V$  is encoded by an  $n$ -bit string  $\omega$ , where  $\omega[i] = 1$  if variable  $i \in S$  and  $\omega[i] = 0$  otherwise. Each processor in the  $k$ -D hypercube is encoded by an  $k$ -bit string  $r$ .

- 1: **for** each  $i \in V$ , compute  $B_i(S)$  and  $A_i(S)$  for all  $S \subseteq V - \{i\}$  by **Algorithm 4**. Each processor  $r$  computes subsets  $S$  s.t.  $r = \omega_S[1, k]$ .
- 2: Compute  $F(S)$  for all  $S \subseteq V$  on  $k$ -D hypercube. Each processor  $r$  computes subsets  $S$  s.t.  $r = \omega_S[1, k]$ .
- 3: Compute  $R(S)$  for all  $S \subseteq V$  on  $k$ -D hypercube. Each processor  $r$  computes subsets  $S$  s.t.  $r = \sim \omega_S[1, k]$ .
- 4: **for** each  $v \in V$  **do**
- 5:     **if**  $v \leq k$  **then**
- 6:         Each processor  $r$  with  $r[v] = 1$  sends all its  $R$  scores to its neighbor  $r' = r \oplus 2^{v-1}$ .
- 7:     **end if**
- 8:     Each processor  $r$  with  $r[v] = 0$  computes  $q_v(S)F(S)R(V - \{v\} - S)$  for all its  $S$ .
- 9:     Compute  $\Gamma_v(G_v)$  for all  $G_v \subseteq V - \{v\}$  with  $|G_v| \leq d$  by **Algorithm 5**.
- 10:    **for** each  $u \in V - \{v\}$  **do**
- 11:       Each processor  $r$  recomputes  $B_v(G_v)$  for  $G_v$  with  $r = \omega_{G_v}[1, k]$ , then adds up all local  $B_v(G_v)\Gamma_v(G_v)$  scores with  $|G_v| \leq d$ .
- 12:       *MPI\_Reduce* is executed on the  $k$ -D hypercube to compute the sum of all  $B_v(G_v)\Gamma_v(G_v)$ , which is  $P(u \rightarrow v, D)$ , obtained on processor  $r$  s.t.  $r[1..k] = 1$ .
- 13:       Processor  $r$  s.t.  $r[1..k] = 1$  evaluates  $P(u \rightarrow v|D) = P(u \rightarrow v, D)/F(V)$ .
- 14:    **end for**
- 15: **end for**

---

### 3.2.3 RUN-TIME AND SPACE COMPLEXITY

We characterize the running time of *ParaREBEL* under the assumption that the maximum indegree  $d$  is a constant.

For any fixed  $i \in V$ , computing  $A_i(L_i)$  for all  $L_i \subseteq V - \{i\}$  takes  $O(2^{n-k} + k(n-k)^d)$  time (**Theorem 1**). Thus, Line 1 takes  $|V| \cdot O(2^{n-k} + k(n-k)^d) = O(n2^{n-k} + kn(n-k)^d)$  time to compute  $B_i, A_i$  scores for all  $i \in V$ .

Line 2 and Line 3 take  $O(n(2^{n-k} + k))$  time each as we pipeline the execution of the  $k$ -D hypercubes in  $2^{n-k} + k$  steps and each step costs  $O(n)$ .

In Line 9, for any  $v \in V$ , computing  $\Gamma_v$  scores takes  $O(2^{n-k})$  time (**Theorem 2**). Line 11 takes  $O(n^d)$  time as there are no more than  $O(n^d)$   $B_v(S)\Gamma_v(S)$  scores on each processor if bounded indegree  $d$  is assumed. In Line 12, *MPI\_Reduce* procedure takes  $O((\tau + \mu m)k)$  time. Thus, the time combined for Lines 4-15 is  $O(((\tau + \mu m)k + n^d)n + 2^{n-k}) \cdot n = O(n2^{n-k})$ .

The total time for overall algorithm is therefore  $O(n2^{n-k} + k(n-k)^d)$ . Given the serial run-time is  $O(n2^n)$ , optimal parallel run-time is  $O(n2^{n-k})$ , which is achieved by our parallel algorithm when  $2^{n-k} > k(n-k)^d$ , i.e.,  $n > k + \log k + d \log(n-k)$ .

Furthermore,  $B, A, \Gamma, F, R$  scores are evenly distributed on the  $2^k$  processors. Therefore, the storage per processor used by the parallel algorithm is  $O(n2^{n-k})$ . Since the space

requirement of the sequential algorithm is  $O(n2^n)$ , our parallel algorithm achieves the optimal space efficiency.

In summary, we obtain the following results.

**Theorem 3** *Algorithm ParaREBEL runs in time  $O(n2^{n-k} + k(n-k)^d)$  and space  $O(n2^{n-k})$ . The algorithm achieves optimal time and space efficiency when  $n > k + \log k + d \log(n - k)$ .*

■

## 4. Experimental Validation

In this section, we present the experiments for evaluating our ParaREBEL algorithm.

### 4.1 Implementation and Computing Environment

We implemented the proposed ParaREBEL algorithm<sup>6</sup> in C++ and MPI and demonstrated its scalability on TACC Stampede<sup>7</sup>, a Dell PowerEdge C8220 cluster. Each compute node in the cluster consists of two Xeon Intel 8-Core E5-2680 processors (16 cores in all), sharing 32 GB memory. All experiments were run with one MPI process per core. To allow more memory per process, only 8 cores in each node were recruited so that each process could use up to 4 GB memory. The maximum number of nodes/cores allowed for a regular user on TACC Stampede is 256/4096. To maintain 4 GB per core, we can only use up to 2048 cores. Thus, all the following experiments were done on up to 2048 cores. We compared our implementation with REBEL<sup>8</sup>, a C++ implementation of the state-of-the-art serial algorithm in (Koivisto, 2006).

We generated a set of synthetic data sets with discrete variables. All datasets contain 500 samples. For each data set, we ran the serial algorithm and our ParaREBEL algorithm to compute the posterior probabilities for all  $n(n-1)$  potential edges. We did two tests: one with varying bounded indegree  $d$  and fixed number of variables  $n$ , the other with varying number of variables  $n$  and fixed bounded indegree  $d$ . In both tests, the total run-times were recorded. We then computed the *speedup* as the ratio between the serial run-time and parallel run-time. We also computed the *efficiency* as the ratio between the serial run-time and the product of number of processors used and the parallel run-time. In the second test, the memory usages per processor were collected and the total memory usages were calculated.

### 4.2 Experimental Results

In the first test, we fixed  $n = 25$  and studied the performance of ParaREBEL algorithm with respect to the bounded indegree  $d$ . The run-times are presented in Table 1. The corresponding speedups and efficiencies are illustrated in Figure 5. Generally, we observed overall good scaling (see speedup plot in Figure 5) for all values of  $d$ . Larger  $d$  increases the computational complexity but leaves the communication unaffected. Thus, the computation is more likely to dominate the communication when  $d$  is increased. This is consistent with the observation that the larger  $d$  scales better than smaller  $d$  (see speedup plot in Figure 5).

6. ParaREBEL is available for download at <http://www.cs.iastate.edu/~yetianc/software.html>.

7. <http://www.tacc.utexas.edu/resources/hpc/stampede>

8. <http://www.cs.helsinki.fi/u/mkhkoivi/REBEL>

As a result, larger  $d$  has better efficiency than smaller  $d$  (see efficiency plot in Figure 5). For  $d = 4$ , the efficiencies are maintained above 0.53 with up to 2048 cores.<sup>9</sup>

Table 1: Run-time for the test data with  $n = 25$  with varying bounded indegree  $d$ .

No.CPUs	Run-time (seconds)	
	$d = 2$	$d = 4$
Serial	1319	2295
4	1284	1330
8	575	594
16	327	338
32	139	146
64	59.9	64.2
128	26.6	29.4
256	11.7	13.8
512	5.2	6.8
1024	2.5	3.6
2048	1.5	2.1

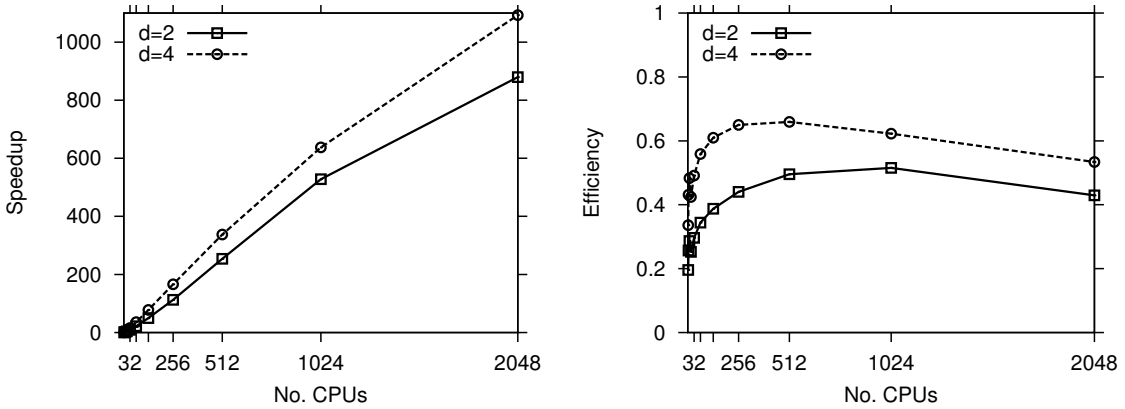


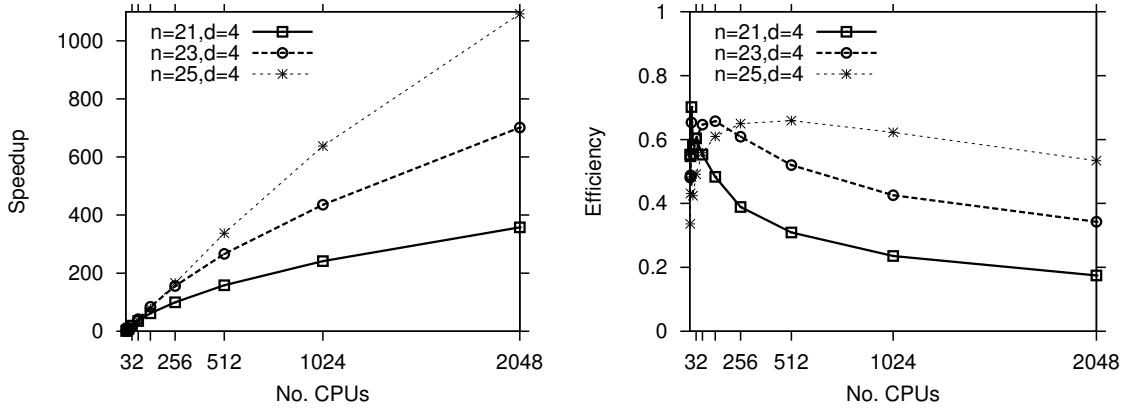
Figure 5: Speedup and Efficiency for the test data with  $n = 25$  with varying bounded indegree  $d$ .

In the second test, we fixed  $d = 4$  and studied the performance of the algorithm with respect to the number of variables ( $n = 21, 23, 25, 27, 29, 31, 33$ ). We first compared the run-times. As showed in Table 2, the run-times are reflective of the exponential dependence on  $n$ . Further, we observed that the algorithm scaled much better on larger  $n$  (Figure 6). This can be explained by that computation complexity increases faster than communication does as  $n$  increases. For  $n = 25$ , the parallel aglorithm maintains an efficiency of about 0.6 with up to 2048 cores. For  $n = 33$ , the problem can only be solved on 1024 and 2048 cores

9. Generally, parallel algorithms with  $efficiency \geq 0.5$  are considered to be successfully parallelized.

Table 2: Run-time for the test data with  $n = 21, 23, 25, 27, 29, 31, 33$  with fixed  $d = 4$ .

No.CPUs	Run-time (seconds)						
	$n = 21$	$n = 23$	$n = 25$	$n = 27$	$n = 29$	$n = 31$	$n = 33$
Serial	96.5	492	2295	-	-	-	-
4	44.1	252	1330	-	-	-	-
8	17.2	94.2	594	-	-	-	-
16	10.3	55.5	338	-	-	-	-
32	5.0	25.5	146	682	-	-	-
64	2.7	11.9	64.2	385	2201	-	-
128	1.6	5.8	29.4	167	864	-	-
256	0.97	3.2	13.8	73.5	389	2540	-
512	0.61	1.8	6.8	33.9	196	987	-
1024	0.4	1.1	3.6	15.9	87	488	2884
2048	0.27	0.7	2.1	7.8	39	215	1452


 Figure 6: Speedup and Efficiency for the test datas with  $n = 21, 23, 25$ .

due to memory constraint. We had a try on  $n = 34$  using 2048 cores but were not able to solve it as it ran out of memory.

One interesting observation is that for any fixed  $d$  and  $n$ , the parallel efficiency increases as the No.CPUs increases, peaks at somewhere in between, then gradually decreases as No.CPUs goes up to 2048 CPUs (See efficiency plot in Figure 6). Mathematically, this optimum can be found by maximizing  $efficiency = 1/(1 + k2^k(n - k)^d)$ , i.e., minimizing  $k2^k(n - k)^d$  over  $k$ . Solving this optimization problem yields  $k^* = n(\ln 2 + 1)/(\ln 2 + 1 + d) \approx 1.7n/(d + 1.7)$ . Plugging in  $n = 25$  and  $d = 4$  yields  $k^* = 7.5 \approx 8$ , i.e.,  $2^{k^*} \approx 256$  cores. Plugging in  $n = 23$  and  $d = 4$  yields  $k^* = 6.8 \approx 7$ , i.e.,  $2^{k^*} \approx 128$  cores. All these results perfectly consist with the observations in Figure 6. This provides a piece of solid experimental evidence for **Theorem 3**. Further, this optimum  $k^*$  is proportional to  $n$ , i.e., the optimal efficiency is achieved by using larger number of cores when problem

Table 3: Memory usage for the test data with  $n = 23, 25, 27, 29, 31, 33$  with fixed  $d = 4$ .

No.CPUs	Memory Usage					
	$n = 23$	$n = 25$	$n = 27$	$n = 29$	$n = 31$	$n = 33$
4	1.88 (481)	8.00 (2049)	-	-	-	-
8	1.88 (240)	8.00 (1025)	-	-	-	-
16	1.88 (121)	8.01 (513)	-	-	-	-
32	2.17 ((70)	8.30 (266)	34.32 (1098)	-	-	-
64	2.49 (40)	8.62 (138)	34.64 (554)	144.68 (2315)	-	-
128	3.31 (27)	9.46 (76)	35.48 (284)	145.53 (1164)	-	-
256	4.93 (20)	11.06 (44)	37.09 (148)	147.07 (588)	606.13 (2425)	-
512	8.40 (17)	14.73 (30)	40.76 (82)	150.87 (302)	615.03 (1230)	-
1024	17.58 (18)	23.62 (24)	49.72 (50)	159.87 (160)	623.88 (624)	2520 (2520)
2048	41.08 (21)	47.32 (24)	72.97 (36)	183.69 (92)	647.27 (324)	2560 (1300)

(The term outside the parentheses is the total memory usage measured in GiB, the term inside the parentheses is the memory usage per core measured in MiB.)

becomes larger. This suggests our ParaREBEL algorithm scales quite well with respect to the problem size  $n$ .

We then examined the actual memory usages with respect to the number of variables  $n$  and the number of cores  $2^k$  in Table 3. For  $n = 23$ , the total memory usage remains the same (1.88 GiB) for  $2^k = 4, 8, 16$  cores, but starts to increase as the number of cores increases from 16 to 2048. This increase is dramatic for the number of cores ranging from 256 to 2048, i.e, the memory usage is doubled when the number of cores is doubled. This can be explained by examining the memory usage per core. For  $2^k = 4, 8, 16$ , the memory usage per core decreases by half when the number of cores is doubled. This is consistent with our theoretical analysis that the space complexity is  $O(n2^{n-k})$  per core. When  $2^k \geq 16$ , the reduction slows down and the memory usage plateaus at about 20 MiB per core. It is speculated that in addition to the memory allocated for storing the  $B, A, F, R, \Gamma$  scores, each core requires extra  $10 \sim 20$  MiB memory to store program execution related data in order to run the program. This overhead is underappreciated when the memory usage per core is dominated by the scores but comes into play in the other case. For  $n = 25$ , total memory usage stays at about 8 GiB for  $2^k = 4 \sim 64$  and starts to increase thereafter; for  $n = 27$ , total memory usage stays at about 35 GiB for  $2^k = 32 \sim 256$  and starts to increase thereafter; for  $n = 29, 31, 33$ , the memory usage per core is dominated by the scores, thus, the total memory usage stays roughly constant with respect to the number of cores examined. Further, it is easily observed that the memory usages (total memory usage and memory usage per core) are reflective of the exponential dependence on  $n$ . Thus, the observations on the memory usage are consistent with our previous analysis of the space complexity.

Moreover, from this test, we observed that it requires at least 4 GB memory per core if  $n - k \geq 23$ . To solve a problem of  $n \geq 34$ , we need 2048 cores with more than 4 GB memory per core or 4096 cores with more than 2 GB memory per core. However, these

resources are unavailable to a regular user on TACC Stampede. Further, we observed that the problem of  $n = 33$  could be solved on 1024 cores in less than one hour, and 2048 cores in less than half an hour. The computing times are still far away from the practical limit. Thus, memory requirement is still the bottleneck that determines the feasibility limit in practice.

## 5. Discussions and Conclusions

Exact Bayesian structure discovery in Bayesian networks requires exponential time and space. In this work, we have presented a parallel algorithm capable of computing the exact posterior probabilities for all  $n(n-1)$  potential edges with optimal time and space efficiency. To our knowledge, this is the first practical parallel algorithm for computing the exact posterior probabilities of structural features in BNs. We demonstrated its capability on datasets with up to 33 variables and its scalability on up to 2048 processors. To our knowledge, 33-variable network is the largest problem solved so far.

In addition, our algorithm makes twofold algorithmic contributions. First, it presents a delicate way to coordinate the computations of correlated DP procedures such that large amount of data exchange is suppressed during the transitions among these DP procedures. Second, it involves the development of two parallel techniques for computing two variants of well-known *zeta transform*. These features or ideas can be extended and applied in developing parallel algorithms for related DP problems. For example, the unbiased algorithm in (Tian and He, 2009) involves similar steps and transforms, thus can be parallelized in similar way. Further, as zeta transforms are fundamental objects in combinatorics and algorithmics, the parallel techniques developed here would also benefit the researches beyond the context of Bayesian networks (Björklund et al., 2007, 2010; Nederlof, 2009).

From the experiments, we observed that memory requirement reached the limit much faster than computing time did. Thus, one of the future work is to improve the algorithm such that less space is used. Particularly, it has a possibility to combine the present algorithm with the method of (Parviainen and Koivisto, 2010) to trade space against time.

ParaREBEL is available at <http://www.cs.iastate.edu/~yetianc/software.html>.

## Acknowledgments

We would like to acknowledge support from the Department of Computer Science at Iowa State University.

## References

- Grama Ananth, Gupta Anshul, Karypis George, and Kumar Vipin. *Introduction to Parallel computing*. Boston, MA: Addison-Wesley, 2003.
- Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Fourier meets möbius: fast subset convolution. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 67–74. ACM, 2007.

- Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Trimmed moebius inversion and graphs of bounded degree. *Theory of Computing Systems*, 47(3):637–654, 2010.
- David M Chickering, Dan Geiger, and David Heckerman. Learning Bayesian networks: Search methods and experimental results. In *Proceedings of the Fifth International Workshop on Artificial Intelligence and Statistics*, pages 112–128, January 1995.
- Gregory F Cooper and Edward Herskovits. A bayesian method for the induction of probabilistic networks from data. *Machine learning*, 9(4):309–347, 1992.
- William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Access Online via Elsevier, 2004.
- Nir Friedman and Daphne Koller. Being bayesian about network structure. a bayesian approach to structure discovery in bayesian networks. *Machine learning*, 50(1-2):95–125, 2003.
- David Heckerman, Christopher Meek, and Gregory Cooper. A bayesian approach to causal discovery. Technical report, MSR-TR-97-05, Microsoft Research, 1997.
- Robert Kennes. Computational aspects of the mobius transformation of graphs. *Systems, Man and Cybernetics, IEEE Transactions on*, 22(2):201–223, 1992.
- Mikko Koivisto. Advances in exact bayesian structure discovery in bayesian networks. In *Proceedings of the 22nd Conference in Uncertainty in Artificial Intelligence*, 2006.
- Mikko Koivisto and Kismat Sood. Exact bayesian structure discovery in bayesian networks. *The Journal of Machine Learning Research*, 5:549–573, 2004.
- Peter KK Loh, Wen-Jing Hsu, and Yi Pan. The exchanged hypercube. *Parallel and Distributed Systems, IEEE Transactions on*, 16(9):866–874, 2005.
- Brandon M Malone, Changhe Yuan, and Eric A Hansen. Memory-efficient dynamic programming for learning optimal bayesian networks. In *AAAI*, 2011.
- Jesper Nederlof. Fast polynomial-space algorithms using möbius inversion: Improving on steiner tree and related problems. In *Automata, Languages and Programming*, pages 713–725. Springer, 2009.
- Olga Nikolova, Jaroslaw Zola, and Srinivas Aluru. A parallel algorithm for exact bayesian network inference. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 342–349. IEEE, 2009.
- Olga Nikolova, Jaroslaw Zola, and Srinivas Aluru. Parallel globally optimal structure learning of bayesian networks. *Journal of Parallel and Distributed Computing*, 2013.
- Sascha Ott, Seiya Imoto, and Satoru Miyano. Finding optimal models for small gene networks. In *Pacific symposium on biocomputing*, volume 9, pages 557–567, 2004.



- Pekka Parviainen and Mikko Koivisto. Exact structure discovery in bayesian networks with less space. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 436–443. AUAI Press, 2009.
- Pekka Parviainen and Mikko Koivisto. Bayesian structure discovery in bayesian networks with less space. In *International Conference on Artificial Intelligence and Statistics*, pages 589–596, 2010.
- Tomi Silander and Petri Myllymäki. A simple approach for finding the globally optimal bayesian network structure. In *Proceedings of the 22th Conference on Uncertainty in Artificial Intelligence*, pages 445–452, 2006.
- Ajit P Singh and Andrew W Moore. Finding optimal bayesian networks by dynamic programming. Technical report, CMU-CALD-05-106, Carnegie Mellon University, 2005.
- Yoshinori Tamada, Seiya Imoto, and Satoru Miyano. Parallel algorithm for learning optimal bayesian network structure. *Journal of Machine Learning Research*, 12:2437–2459, 2011.
- Jin Tian and Ru He. Computing posterior probabilities of structural features in bayesian networks. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 538–547. AUAI Press, 2009.